CLI/ODBC programming

DB2 Information Management Software

http://www-136.ibm.com/developerworks/db2

Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Before you start

What is this tutorial about?

This tutorial introduces you to CLI/ODBC programming and walks you through the basic steps used to construct a CLI/ODBC application. It also introduces you to the process to convert one or more high-level programming language source code files containing CLI/ODBC function calls into an executable application. In this tutorial, you will learn:

- What environment, connection, statement, and descriptor handles are, and how they are used by CLI/ODBC applications
- The steps involved in developing a CLI/ODBC application
- The correct sequence to use when calling CLI/ODBC functions
- How to establish a database connection from a CLI/ODBC application
- How to configure a CLI/ODBC driver
- How to evaluate CLI/ODBC function return codes and obtain diagnostic information when errors occur
- How to convert source code files containing CLI/ODBC function calls into an executable application

This is the fourth in a series of seven tutorials that you can use to help prepare for the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703). The material in this tutorial primarily covers the objectives in Section 4 of the exam, entitled "ODBC/CLI programming." You can view these objectives at: *http://www.ibm.com/certify/tests/obj703.shtml*.

You do not need a copy of DB2 Universal Database to complete this tutorial. However, you can download a free trial version of *IBM DB2 Universal Database* Enterprise Server Edition for reference.

Who should take this tutorial?

To take the DB2 UDB V8.1 Family Application Development exam, you must have already passed the DB2 UDB V8.1 Family Fundamentals exam (Exam 700). You can use the DB2 Family Fundamentals tutorial series (see Resources on page 33) to prepare for that test. It is a very popular tutorial series that has helped many people understand the fundamentals of the DB2 family of products.

Although not all materials discussed in the Family Fundamentals tutorial series are required to understand the concepts described in this tutorial, you should have a basic knowledge of:

- DB2 instances

- Databases
- Database objects
- DB2 security

This tutorial is one of the tools that can help you prepare for Exam 703. You should also take advantage of one or more of the Resources on page 33 identified at the end of this tutorial for more information.

About the author

Roger E. Sanders is a database performance engineer with Network Appliance, Inc. He has been designing and developing database applications for more than 18 years and he is the author of eight books on DB2 Universal Database, including *DB2 Universal Database v8.1 Certification Exam 703 Study Guide*, *DB2 Universal Database v8.1 Certification Exams 701 and 706 Study Guide*, and *DB2 Universal Database v8.1 Certification Exam 700 Study Guide*. In addition, Roger is a regular contributor to *DB2 Magazine* and he frequently presents at International DB2 User's Group (IDUG) and regional DB2 User's Group (RUG) conferences. Roger holds eight IBM DB2 certifications, including:

- IBM Certified Advanced Database Administrator -- DB2 Universal Database V8.1 for Linux, UNIX, and Windows
- IBM Certified Database Administrator -- DB2 Universal Database V8.1 for Linux, UNIX, and Windows
- IBM Certified Application Developer -- DB2 Universal Database V8.1 Family
- IBM Certified Database Associate -- DB2 Universal Database V8.1 Family.
- IBM Certified Advanced Technical Expert -- DB2 for Clusters

You can reach Roger at *rsanders@netapp.com*.

Notices and trademarks

Section 2. Introduction to CLI/ODBC programming

What is CLI/ODBC?

One of the biggest drawbacks to developing applications with embedded SQL is the lack of interoperability that such an application affords: Embedded SQL applications developed specifically for DB2 UDB will have to be modified (and in some cases completely rewritten) before they can interact with other relational database management systems (RDBMSs). Because this limitation exists in every embedded SQL application, regardless of the RDBMS for which it is written, in the early 1990s the X/Open Company and the SQL Access Group (SAG), now a part of X/Open, jointly developed a standard specification for a callable SQL interface. This interface was known as the *X/Open Call-Level Interface,* or *X/Open CLI.* Much of the X/Open CLI specification was later accepted as part of the ISO CLI international standard. The primary purpose of X/Open CLI was to increase the portability of database applications by allowing them to become independent of any one database management system's programming interface.

In 1992, Microsoft developed a callable SQL interface known as *Open Database Connectivity* (or *ODBC*) for the Microsoft Windows operating system. ODBC is based on the X/Open CLI standards specification but it provides extended functionality and capability that is not part of X/Open CLI. ODBC relies on an operating environment in which data source-specific ODBC drivers are dynamically loaded at application run time (based on information provided when a connection is requested) by a component known as the *ODBC Driver Manager.* Each data source-specific driver is responsible for implementing any or all of the functions defined in the ODBC specification, and for providing interaction with the data source for which the driver was written. The ODBC Driver Manager provides a central point of control; as an ODBC application executes, each ODBC function call it makes is sent to the ODBC Driver Manager, where it is forwarded to the appropriate data source driver for processing. By using drivers, an application can be linked directly to a single ODBC driver library, rather than to each product-specific database itself.

DB2's *Call Level Interface* (also known as *DB2 CLI*) is based on the ISO CLI international standard. It provides most of the functionality that is found in the ODBC specification. Applications that use DB2 CLI instead of ODBC are linked directly to the DB2 CLI load library, and any ODBC Driver Manager can load this library as an ODBC driver. DB2 UDB applications can also use the DB2 CLI load library independently. However, when the library is used in this manner, the application itself will not be able to communicate with other data sources.

Differences between embedded SQL and CLI/ODBC

In the third tutorial in this series (see Resources on page 33), we saw that embedded SQL applications are constructed by embedding SQL statements

directly into one or more source code files that are written using a high-level programming language. CLI/ODBC applications, on the other hand, rely on a standardized set of *Application Programming Interface* (API) functions to send SQL statements to the DB2 Database Manager for processing. Embedded SQL applications and CLI/ODBC applications also differ in the following ways:

- CLI/ODBC applications do not require the explicit declaration and use of host variables; any variable can be used to send data to or retrieve data from a data source.
- Cursors do not have to be explicitly declared by CLI/ODBC applications. Instead, cursors are automatically generated, if needed, whenever the `SQLExecute()` function or the `SQLExecDirect()` function are executed. (We'll take a closer look at these two functions a little later.)
- Cursors do not have to be explicitly opened in CLI/ODBC applications; cursors are implicitly opened as soon as they are generated.
- CLI/ODBC functions manage environment, connection, and SQL statement-related information using *handles*. This technique allows such information to be treated as abstract objects. The use of handles eliminates the need for CLI/ODBC applications to use database product-specific data structures, such as DB2 UDB's SQLCA and SQLDA data structures. (We discussed these structures in detail in the third tutorial in this series, see Resources on page33).
- Unlike embedded SQL, CLI/ODBC applications inherently have the ability to establish multiple connections to multiple data sources, or to the same data source, at the same time. (Embedded SQL applications can only connect to multiple data sources at the same time if Type 2 connections are used.)

Despite these differences, there is one important concept common to both embedded SQL applications and CLI/ODBC applications: CLI/ODBC applications can execute any SQL statement that can be dynamically prepared in an embedded SQL application. This is guaranteed because CLI/ODBC applications pass all of their SQL statements directly to the data source for dynamic execution. (CLI/ODBC applications can also execute some SQL statements that cannot be dynamically prepared, such as compound SQL statements, but for the most part, static SQL is not supported.)

Because the data source processes all SQL statements submitted by a CLI/ODBC application, the portability of CLI/ODBC applications is guaranteed. This is not always the case with embedded SQL applications, since the way SQL statements are dynamically prepared can vary with each relational database product used. Also, because `COMMIT` and `ROLLBACK` SQL statements can be dynamically prepared by some database products (including DB2 UDB) but not by others, they are typically not used in CLI/ODBC applications. Instead, CLI/ODBC applications rely on the `SQLEndTran()` function to terminate active transactions (when manual commit is used). This ensures that CLI/ODBC applications can successfully end transactions, regardless of the database product being used.

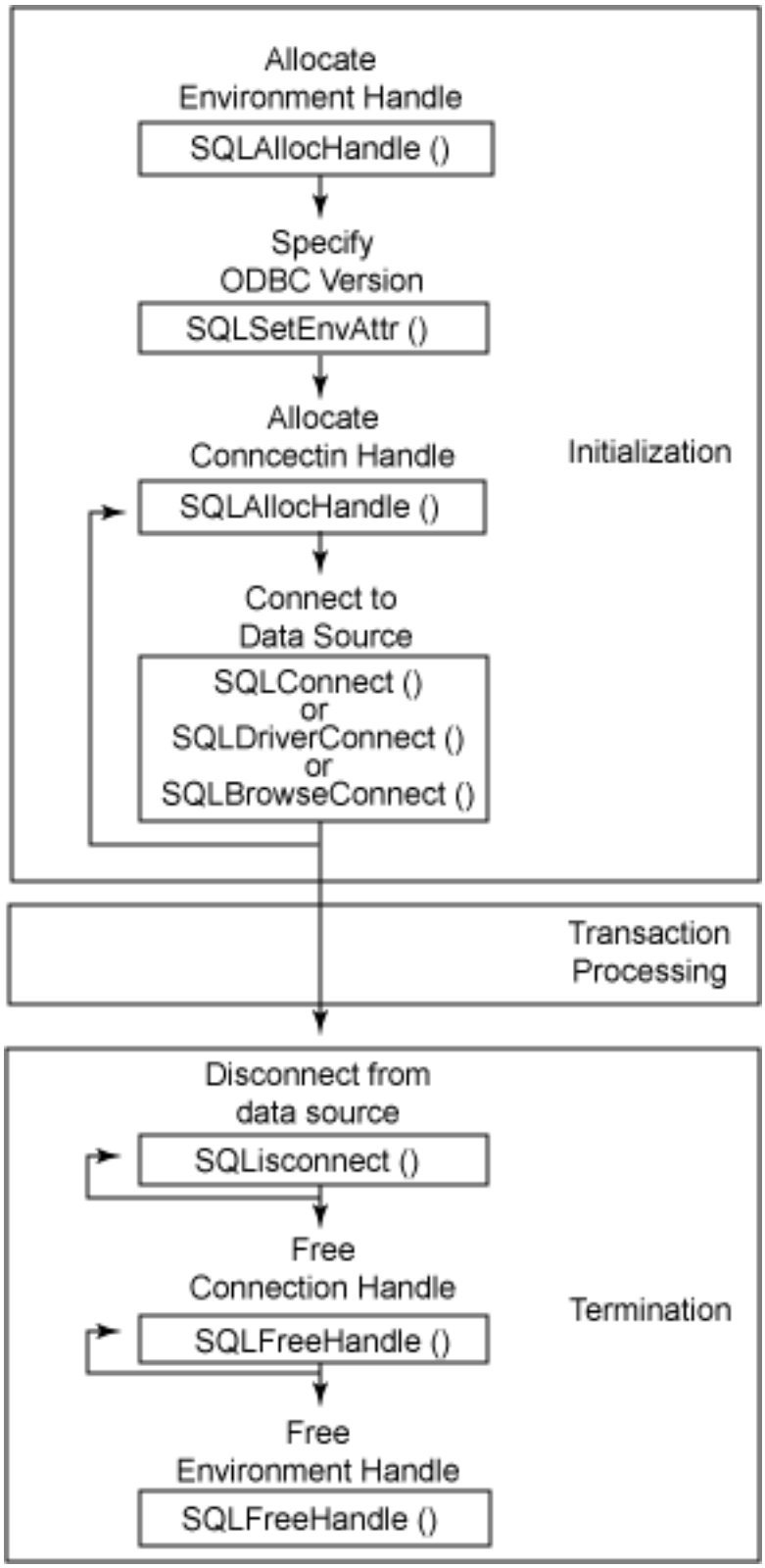
Section 3. Constructing an CLI/ODBC application

Parts of a CLI/ODBC application

All CLI/ODBC applications are constructed such that they perform three distinct tasks:

- Initialization
- Transaction processing
- Termination

The work associated with these three tasks is conducted by invoking one or more CLI/ODBC functions. Furthermore, many of the CLI/ODBC functions used to carry out these tasks must be called in a specific order or an error will occur. The following illustration identifies some of the basic CLI/ODBC functions that are used to perform initialization and termination.



CLI/ODBC applications can perform tasks other than the three outlined above, such as error handling and message processing. We'll look at how errors are handled in a CLI/ODBC application in Diagnostics and error handling on page ?.

Allocating resources

During initialization, resources that will be needed to process transactions are allocated (and initialized) and connections to any data source(s) with which the transaction processing task will interact are established. The resources used by CLI/ODBC applications consist of special data storage areas that are identified by unique *handles.* (A handle is simply a pointer variable that refers to a data object controlled by DB2 CLI or the ODBC Driver Manager and referenced by CLI/ODBC function calls.) By using data storage areas and handles, CLI/ODBC applications are freed from the responsibility of allocating and managing global variables and/or data structures like the SQLCA and SQLDA data structures that are used with embedded SQL applications. Four different types of handles are available:

- *Environment handles:* A pointer to a data storage area that contains CLI/ODBC-specific information that is global in nature.
- *Connection handles:* A pointer to a data storage area that contains information about a data source (database) connection managed by CLI/ODBC.
- *Statement handles:* A pointer to a data storage area containing specific information about a single SQL statement.
- *Descriptor handles:* A pointer to a data storage area that contains a collection of metadata that describes either the application variables that have been bound to parameter markers in an SQL statement or the application variables that have been bound to the columns of a result data set that was produced in response to a query.

Every CLI/ODBC application must begin by allocating an environment handle. Only one environment handle is usually allocated per application, and that handle must exist before any other handles can be allocated; all other handles are managed within the context of the environment handle used. Environment handles are allocated by calling the `SQLAllocHandle()` function with the `SQL_HANDLE_ENV` option specified. The source code used to allocate an environment handle looks something like:

```
SQLHANDLE  EnvHandle = 0;
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &EnvHandle);
```

With CLI/ODBC applications, connections to data sources are made via connection handles. Therefore, a connection handle must exist before a connection to any data source can be established. Connection handles are allocated by calling the `SQLAllocHandle()` function with the `SQL_HANDLE_DBC` option and a valid environment handle specified. The source code used to allocate a connection handle looks something like:

```
SQLHANDLE  ConHandle = 0;
```

```
if (EnvHandle != 0)
    SQLAllocHandle(SQL_HANDLE_DBC, EnvHandle, &ConHandle);
```

The statement handle is the real workhorse of CLI/ODBC. Statement handles are used to process each SQL statement in an application (both user-defined SQL statements and SQL statements that are performed behind the scenes when certain CLI/ODBC functions are called). Notably, statement handles are used to:

- Bind application variables to parameter markers used in a statement
- Prepare and submit a statement to the appropriate data source for execution
- Obtain metadata about any result data set(s) produced in response to a statement
- Bind application variables to columns found in any result data set(s) produced
- Retrieve (fetch) data from any result data set(s) produced
- Obtain diagnostic information when a statement fails to execute

Each SQL statement used in an application must have its own statement handle, and each statement handle used can only be associated with one connection handle. However, any number of statement handles can be associated with a single connection handle. Statement handles are allocated by calling the `SQLAllocHandle()` function with the `SQL_HANDLE_STMT` option and a valid connection handle specified. Thus, the source code used to allocate a statement handle looks something like:

```
SQLHANDLE  StmtHandle = 0;
if (ConHandle != 0)
    SQLAllocHandle(SQL_HANDLE_STMT, ConHandle, &StmtHandle);
```

Whenever a statement handle is allocated, four descriptor handles are automatically allocated and associated with the statement handle as part of the allocation process. Once allocated, these descriptor handles remain associated with the corresponding statement handle until it is destroyed (at which time the descriptor handles will also be destroyed). Most CLI/ODBC operations can be performed using these implicitly defined descriptor handles. However, it is possible to explicitly allocate one or more descriptor handles by calling the `SQLAllocHandle()` function with the `SQL_HANDLE_DESC` option and a valid connection handle specified.

Declaring the application CLI/ODBC version

Both CLI and ODBC use product-specific drivers to communicate with data sources (databases), and most of these drivers contain a set of dynamic parameters that can be changed to alter the driver's behavior to meet an application's needs. These parameters are referred to as *attributes,* and every

environment, connection, and statement handle allocated has its own set of attributes. We'll look at the CLI/ODBC functions that are used to retrieve and/or change the attributes associated with an environment, connection, or statement handle a little later. One such environment attribute is the `SQL_ATTR_ODBC_VERSION` attribute, which must be assigned the value `SQL_OV_ODBC3` or `SQL_OV_ODBC2` after an environment handle has been allocated but before any corresponding connection handles are allocated. This tells DB2 CLI and/or the ODBC Driver Manager that the application intends to adhere to the CLI/ODBC 3.x (or later) specification or the CLI/ODBC 2.0 (or earlier) specification, respectively. The source code used to tell DB2 CLI or the ODBC Driver Manager that an application intends to adhere to the CLI/ODBC 3.x (or later) specification looks something like:

```
SQLSetEnvAttr(EnvHandle, SQL_ATTR_ODBC_VERSION,
    (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
```

It is important that DB2 CLI and/or the ODBC Driver Manager knows the specification for which an application has been coded, because many of the return code values (otherwise known as SQLSTATEs; we'll discuss these in more detail in SQLSTATEs on page 28) that are returned by a CLI/ODBC function vary from one version to the next. Additionally, later versions of DB2 CLI and ODBC allow wild cards to be used in some function parameters, while earlier versions do not.

Establishing a data source connection

Earlier, we saw that in order to perform any type of operation against a database, a connection to that database must first be established. With CLI/ODBC applications, three different functions can be used to establish a data source connection. They are:

- `SQLConnect()`
- `SQLDriverConnect()`
- `SQLBrowseConnect()`

Applications can use any combination of these functions to connect to any number of data sources, although some data sources may limit the number of active connections they support. (An application can find out how many active connections a particular data source supports by calling the `SQLGetInfo()` function with the `SQL_MAX_DRIVER_CONNECTIONS` information type specified.)

The `SQLConnect()` function is by far the simplest CLI/ODBC connection function available. When used, the `SQLConnect()` function assumes that the only information needed to establish a connection is a data source name and, optionally, a user ID (authorization ID) and password. (Any other information needed is stored in either the `[COMMON]` section of the `db2cli.ini` file, the `[ODBC]` section of the `ODBC.INI` file, or the ODBC subkey in the system

registry.) This function works well for applications that need to connect to data sources that only require a user ID and password, and for applications that want to provide their own connection interface or that require no user interface at all.

The `SQLDriverConnect()` function allows an application to send connection information to a data source driver using a connection string (as opposed to storing this information in the `db2cli.ini` file, the `ODBC.INI` file, or the system registry and allowing the driver to retrieve it). A connection string is simply a series of keyword/value pairs, separated by semicolons, that contain information that is to be used to establish a connection to a data source. The table below outlines some of the more common keyword/value pairs.

Keyword/Value	**Purpose**
`DSN=`*`DataSourceName`*	Specifies the name of a data source (as returned by the `SQLDataSources()` function) that a connection is to be established with.
`UID=`*`UserID`*	Specifies the user ID (authorization ID) of the user attempting to establish the connection.
`PWD=`*`Password`*	Specifies the password corresponding to the user ID (authorization ID) specified. If a password is not required for the specified user ID, an empty password string (`PWD=;`) should be used.
`NEWPWD=`*`NewPassword`*	Specifies the new password that is to be assigned to the user ID (authorization ID) specified. If the `NEWPWD` keyword is used but no new password is provided (`NEWPWD=;`), the DB2 CLI driver will prompt the user to provide a new password.

Imagine that we have an application that always connects to a database named PAYROLL using the authorization ID `db2admin` and the corresponding password `ibmdb2`. We could use a connection string with the `SQLDriverConnect()` function that looks something like:

```
DSN=PAYROLL;UID=db2admin;PWD=ibmdb2;
```

When invoked, the `SQLDriverConnect()` function parses the connection string provided and, using the data source name specified, attempts to retrieve additional information needed from the system to establish a connection. Using this information, the function then logs on to the appropriate server and attempts to connect to the designated data source.

Applications using the `SQLDriverConnect()` function can also let the driver prompt the user for any connection information needed. For example, when the `SQLDriverConnect()` function is called with an empty connection string, DB2 CLI will display a dialog that looks something like:

This dialog prompts the user to select a data source from a list of data sources recognized by DB2 CLI, and to provide a user ID along with a corresponding password. Once this information has been provided, an appropriate connection string is constructed and used to establish a connection to the appropriate data source. Whether this dialog will be displayed is determined by one of the parameter values passed to the `SQLDriverConnect()` function: If this function is called with the `SQL_DRIVER_PROMPT`, `SQL_DRIVER_COMPLETE`, or `SQL_DRIVER_COMPLETE_REQUIRED` option specified, the dialog will be displayed if the connection string provided does not contain enough information to establish a data source connection. On the other hand, if this function is called with the `SQL_DRIVER_NOPROMPT` option specified and more information is needed, an error will be generated.

Like the `SQLDriverConnect()` function, the `SQLBrowseConnect()` function uses a connection string to send connection information to a driver. However, unlike the `SQLDriverConnect()` function, the `SQLBrowseConnect()` function can be used to construct a complete connection string at application run time. This difference allows an application to use its own dialog to prompt users for connection information, thereby retaining control over its look and feel. Application-specific dialogs can also search (browse) the system for data sources that can be used by a particular driver, possibly in several steps. For example, an application might first browse the network for servers, and after choosing a server it can then browse the server for databases that can be accessed by a specific driver.

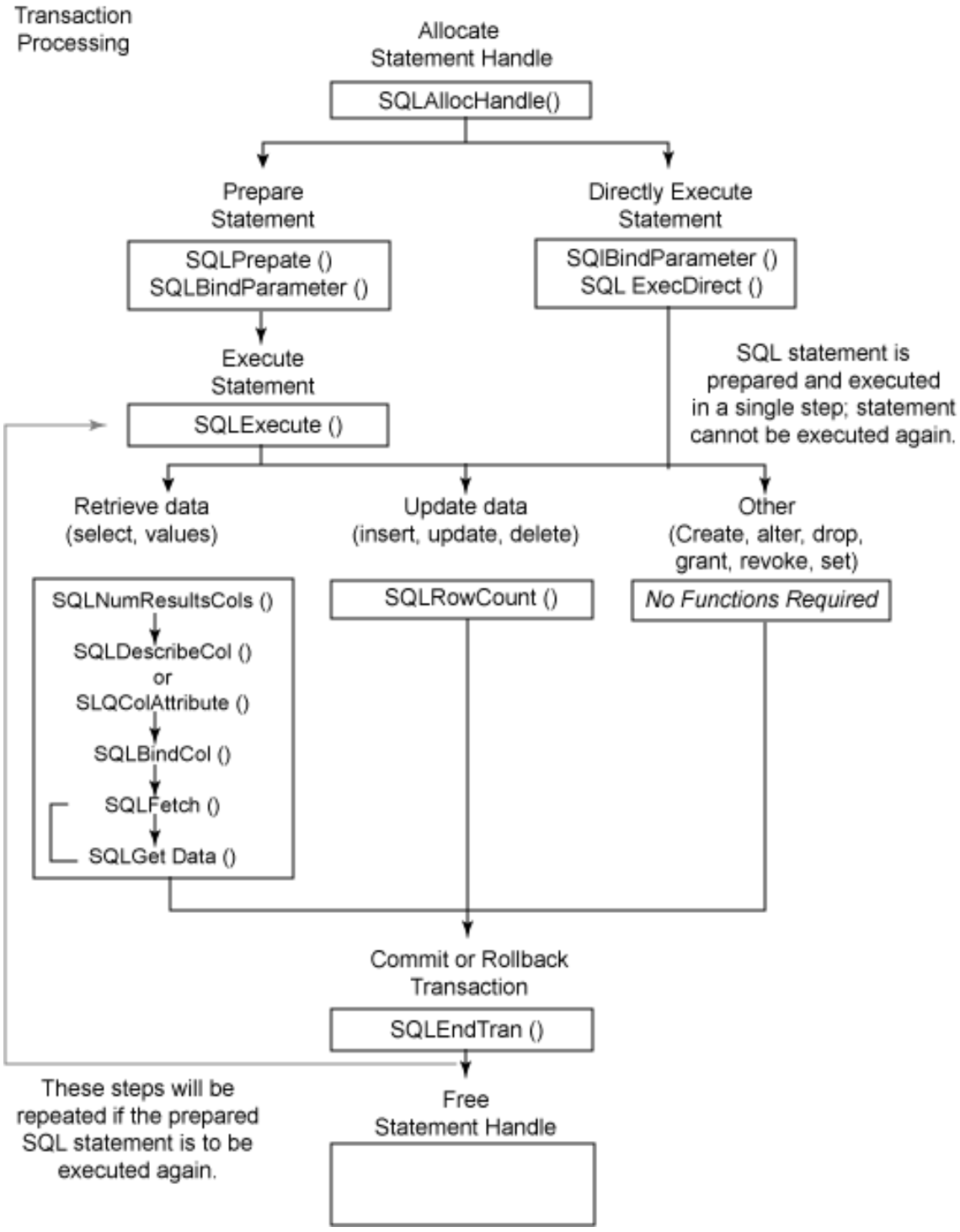
Transaction processing

Once the appropriate initialization has been performed by a CLI/ODBC application, the focus shifts to processing transactions. This is where the SQL statements that query and/or manipulate data are passed to the appropriate data source (which in our case is typically a DB2 UDB database) by various CLI/ODBC function calls for processing. During transaction processing, a CLI/ODBC application performs the following five steps, in the order shown:

1. Allocates one or more statement handles
2. Prepares and executes one or more SQL statements
3. Retrieves and processes any results produced
4. Terminates the current transaction by committing it or rolling it back
5. Frees all statement handles allocated

We'll discuss each of these steps in more detail in the next few panels.

The following illustration shows the basic steps that are performed during the transaction processing task and identifies the CLI/ODBC function calls that are typically used to execute each step.



Allocating statement handles

As mentioned earlier, a statement handle refers to a data object that contains information about a single SQL statement. This information includes:

° The text of the SQL statement

- Details about the cursor (if any) that is associated with the statement
- Bindings for all SQL statement parameter marker variables
- Bindings for all result data set column variables
- The statement execution return code
- Status information

And, as we saw in Allocating resources on page 9, statement handles are allocated by calling the `SQLAllocHandle()` function with the `SQL_HANDLE_STMT` option and a valid connection handle specified. At a minimum, one statement handle must be allocated before any SQL statements can be executed by a CLI/ODBC application.

Preparing and executing SQL statements

Once a statement handle has been allocated, there are two ways in which SQL statements used in a CLI/ODBC application can be processed:

- *Prepare and execute*. This approach separates the preparation of the SQL statement from its execution and is typically used when an SQL statement is to be executed repeatedly. This method is also used when an application needs advance information about the columns that will exist in the result data set produced when the SQL statement is executed. The CLI/ODBC functions `SQLPrepare()` and `SQLExecute()` are used to process SQL statements in this manner.
- *Execute immediately*. This approach combines the preparation and the execution of an SQL statement into a single step and is typically used when an SQL statement is to be executed only once. This method is also used when the application does not need additional information about any result data set that may be produced when the SQL statement is executed. The CLI/ODBC function `SQLExecDirect()` is used to process SQL statements in this manner.

Both of these methods allow the use of parameter markers in place of constants and/or expressions in an SQL statement. Parameter markers are represented by the question mark (?) character, and they indicate the position in the SQL statement where the current value of one or more application variables is to be substituted when the statement is executed. When an application variable is associated with a specific parameter marker in an SQL statement, that variable is said to be *bound* to the parameter marker. Such binding is carried out by calling the `SQLBindParameter()` function. Once an application variable is bound to a parameter marker, the association with that variable remains in effect until it is overridden or until the corresponding statement handle is freed. Although binding can take place any time after an SQL statement has been prepared, data is not actually retrieved from a bound variable until the SQL statement to which the application variable has been bound is executed.

The following example, written in the C programming language, illustrates how an application variable would be bound to a parameter marker that has been

coded in a simple SELECT SQL statement. It also illustrates the way in which a value would be provided for the bound parameter before the statement is executed:

```
...
// Define A SELECT SQL Statement That Uses A Parameter Marker
strcpy((char *) SQLStmt, "SELECT EMPNO, LASTNAME FROM ");
strcat((char *) SQLStmt, "EMPLOYEE WHERE JOB = ?");

// Prepare The SQL Statement
RetCode = SQLPrepare(StmtHandle, SQLStmt, SQL_NTS);

// Bind The Parameter Marker Used In The SQL Statement To
// An Application Variable
RetCode = SQLBindParameter(StmtHandle, 1,
              SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
              sizeof(JobType), 0, JobType,
              sizeof(JobType), NULL);

// Populate The "Bound" Application Variable
strcpy((char *) JobType, "DESIGNER");

// Execute The SQL Statement
RetCode = SQLExecute(StmtHandle);
...
```

Retrieving and processing results

Once an SQL statement has been prepared and executed, any results produced will need to be retrieved and processed. (Result information is stored in the data storage areas that are referenced by the connection and statement handles that are associated with the SQL statement that was executed.) If the SQL statement executed was anything other than a SELECT or a VALUES statement, the only additional processing required after execution is a check of the CLI/ODBC function return code to ensure that the statement executed as expected. However, if a SELECT statement or VALUES statement was executed, the following additional steps are needed to retrieve each row of data from the result data set produced:

1. Determine the structure (i.e., the number of columns, column data types, and data lengths) of the result data set produced. This is done by executing the `SQLNumResultCols()`, `SQLDescribeCol()`, and/or the `SQLColAttributes()` functions.
2. Bind application variables to the columns in the result data set using the `SQLBindCol()` function (optional).
3. Repeatedly fetch the next row of data from the result data set produced and copy it to the bound application variables. This is typically done by repeatedly calling the `SQLFetch()` function within a loop. (Values for columns that were not bound to application variables in Step 2 can be retrieved by calling the `SQLGetData()` function each time a new row is

fetched.)

In the first step, the prepared or executed SQL statement is analyzed to determine the structure of the result data set produced. If the SQL statement was hard-coded into the application, this step is unnecessary because the structure of the result data set produced is already known. However, if the SQL statement was generated at application run time, then the result data set produced must be queried to obtain this information.

Once the structure of a result data set is known, one or more application variables can be bound to specific columns in the result data set, just as application variables are bound to SQL statement parameter markers. In this case, application variables are used as output arguments rather than input arguments, and data is retrieved and written directly to them whenever the `SQLFetch()` function is called. However, because the `SQLGetData()` function can also be used to retrieve data from a result data set, application variable/column binding is optional.

In the third step, data stored in the result data set is retrieved by repeatedly calling the `SQLFetch()` function (usually in a loop) until data is no longer available. If application variables have been bound to columns in the result data set, their values are automatically updated each time `SQLFetch()` is called. On the other hand, if column binding was not performed, the `SQLGetData()` function can be used to copy data from a specific column to an appropriate application variable. The `SQLGetData()` function can also be used to retrieve large variable length column data values in several small pieces (which cannot be done when bound application variables are used). All data stored in a result data set can be retrieved using any combination of these two methods.

The following example, written in the C programming language, illustrates how application variables could be bound to the columns in a result data set, and how data in a result data set is normally retrieved by repeatedly calling the `SQLFetch()` function:

```
...
// Bind The Columns In The Result Data Set Returned
// To Application Variables
SQLBindCol(StmtHandle, 1, SQL_C_CHAR, (SQLPOINTER)
    EmpNo, sizeof(EmpNo), NULL);

SQLBindCol(StmtHandle, 2, SQL_C_CHAR, (SQLPOINTER)
    LastName, sizeof(LastName), NULL);

// While There Are Records In The Result Data Set
// Produced, Retrieve And Display Them
while (RetCode != SQL_NO_DATA)
{
    RetCode = SQLFetch(StmtHandle);
    if (RetCode != SQL_NO_DATA)
        printf("%-8s %s\n", EmpNo, LastName);
}
...
```

Managing transactions

You may recall that a *transaction* (also known as a *unit of work*) is a sequence of one or more SQL operations that are grouped together as a single unit, usually within an application process. A transaction is considered to be *atomic* because it is indivisible: either all of its work is carried out or none of its work is carried out. A given transaction can be comprised of any number of SQL operations, from a single operation to many hundreds or even thousands, depending upon what is considered a single step within your business logic. Transactions are important because the initiation and termination of a single transaction defines points of data consistency within a database; either the effects of all operations performed within a transaction are applied to the database and made permanent (committed), or the effects of all operations performed are backed out (rolled back) and the database is returned to the state it was in before the transaction was initiated.

From a transaction processing viewpoint, a CLI/ODBC application can be configured such that it runs in one of two modes: *automatic commit* or *manual commit.* When automatic commit mode is used, each individual SQL statement is treated as a complete transaction, and each transaction is automatically committed after the SQL statement successfully executes. For anything other than `SELECT` SQL statements, the commit operation takes place immediately after the statement is executed. For `SELECT` statements, the commit operation takes place immediately after the cursor being used to process the result data set is closed. (Remember that CLI/ODBC automatically declares and opens a cursor if one is needed.) Automatic commit mode is the default commit mode and is usually sufficient for simple CLI/ODBC applications. However, larger applications, particularly those that perform update operations, should switch to manual commit mode as soon as a data source connection is established. The call performed to use manual commit looks something like:

```
SQLSetConnectAttr(ConHandle, SQL_ATTR_AUTOCOMMIT,
   SQL_AUTOCOMMIT_OFF, SQL_IS_UINTEGER);
```

When manual commit mode is used, transactions are started implicitly the first time an application accesses a data source, and transactions are explicitly ended when the `SQLEndTran()` function is called. This CLI/ODBC function is used to either roll back or commit all changes made by the current transaction. Thus, all operations performed against a data source between the time at which it is first accessed and the time at which the `SQLEndTran()` function is called are treated as a single transaction.

Freeing statement handles

When the results of an SQL statement have been processed and the SQL statement data storage area that was allocated when transaction processing

began is no longer needed, the memory reserved for that data storage should be freed. The data storage area associated with a particular statement handle is freed by calling the `SQLFreeHandle()` function with the `SQL_HANDLE_STMT` option and the appropriate statement handle specified (for example, `SQLFreeHandle(SQL_HANDLE_STMT, StmtHandle)`). When invoked, this CLI/ODBC function performs the following tasks:

- Unbinds all previously bound parameter application variables
- Unbinds all previously bound column application variables
- Closes any cursors that are open and discards their results
- Destroys the statement handle and releases all associated resources

If a statement handle is not freed, it can be used to process other SQL statements.

Terminating data source connections

Just before a CLI/ODBC application terminates and control is returned to the operating system, all data source connections that have been established should be terminated and all resources that were allocated during initialization should be freed. (Usually, these resources consist of an environment data storage area and one or more connection data storage areas.)

Existing database connections are terminated by calling the `SQLDisconnect()` function with the appropriate connection handle specified. Corresponding connection data storage areas are freed by calling the `SQLFreeHandle()` function with the `SQL_HANDLE_DBC` option and the appropriate connection handle specified. Once all previously allocated connection data storage areas have been freed, the environment data storage area with which the connection data storage areas were associated is also freed by calling the `SQLFreeHandle()` function, this time with the `SQL_HANDLE_ENV` option and the appropriate environment handle specified.

Putting it all together

Now that we have examined the basic components that comprise all CLI/ODBC applications, let's see how these components come together to produce a CLI/ODBC application. A simple CLI/ODBC application, written in the C programming language, that obtains and prints employee identification numbers and last names for all employees who have the job title "designer" might look something like:

```
#include <stdio.h>
#include <string.h>
#include <sqlcli1.h>
```

```
int main()
{
    // Declare The Local Memory Variables
    SQLHANDLE  EnvHandle = 0;
    SQLHANDLE  ConHandle = 0;
    SQLHANDLE  StmtHandle = 0;
    SQLRETURN  RetCode = SQL_SUCCESS;

    SQLCHAR    SQLStmt[255];
    SQLCHAR    JobType[10];
    SQLCHAR    EmpNo[10];
    SQLCHAR    LastName[25];

    /*-------------------------------------------------------*/
    /* INITIALIZATION                                        */
    /*-------------------------------------------------------*/

    // Allocate An Environment Handle
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE,
        &EnvHandle);

    // Set The ODBC Application Version To 3.x
    if (EnvHandle != 0)
        SQLSetEnvAttr(EnvHandle, SQL_ATTR_ODBC_VERSION,
            (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);

    // Allocate A Connection Handle
    if (EnvHandle != 0)
        SQLAllocHandle(SQL_HANDLE_DBC, EnvHandle,
            &ConHandle);

    // Connect To The Appropriate Data Source
    if (ConHandle != 0)
        RetCode = SQLConnect(ConHandle, (SQLCHAR *) "SAMPLE",
                      SQL_NTS, (SQLCHAR *) "db2admin",
                      SQL_NTS, (SQLCHAR *) "ibmdb2",
                      SQL_NTS);

    /*-------------------------------------------------------*/
    /* TRANSACTION PROCESSING                                */
    /*-------------------------------------------------------*/

    // Allocate An SQL Statement Handle
    if (ConHandle != 0 && RetCode == SQL_SUCCESS)
        SQLAllocHandle(SQL_HANDLE_STMT, ConHandle,
           &StmtHandle);

    // Define A SELECT SQL Statement That Uses A Parameter
    // Marker
    strcpy((char *) SQLStmt, "SELECT EMPNO, LASTNAME FROM ");
    strcat((char *) SQLStmt, "EMPLOYEE WHERE JOB = ?");

    // Prepare The SQL Statement
    RetCode = SQLPrepare(StmtHandle, SQLStmt, SQL_NTS);

    // Bind The Parameter Marker Used In The SQL Statement To
    // An Application Variable
    RetCode = SQLBindParameter(StmtHandle, 1,
                  SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                  sizeof(JobType), 0, JobType,
                  sizeof(JobType), NULL);
```

```
    // Populate The "Bound" Application Variable
    strcpy((char *) JobType, "DESIGNER");

    // Execute The SQL Statement
    RetCode = SQLExecute(StmtHandle);

    // If The SQL Statement Executed Successfully, Retrieve
    // The Results
    if (RetCode == SQL_SUCCESS)
    {
        // Bind The Columns In The Result Data Set Returned
        // To Application Variables
        SQLBindCol(StmtHandle, 1, SQL_C_CHAR, (SQLPOINTER)
            EmpNo, sizeof(EmpNo), NULL);

        SQLBindCol(StmtHandle, 2, SQL_C_CHAR, (SQLPOINTER)
            LastName, sizeof(LastName), NULL);

        // While There Are Records In The Result Data Set
        // Produced, Retrieve And Display Them
        while (RetCode != SQL_NO_DATA)
        {
            RetCode = SQLFetch(StmtHandle);
            if (RetCode != SQL_NO_DATA)
                printf("%-8s %s\n", EmpNo, LastName);
        }
    }

    // Commit The Transaction
    RetCode = SQLEndTran(SQL_HANDLE_DBC, ConHandle,
                 SQL_COMMIT);

    // Free The SQL Statement Handle
    if (StmtHandle != 0)
        SQLFreeHandle(SQL_HANDLE_STMT, StmtHandle);

    /*-------------------------------------------------------*/
    /* TERMINATION                                           */
    /*-------------------------------------------------------*/

    // Terminate The Data Source Connection
    if (ConHandle != 0)
        RetCode = SQLDisconnect(ConHandle);

    // Free The Connection Handle
    if (ConHandle != 0)
        SQLFreeHandle(SQL_HANDLE_DBC, ConHandle);

    // Free The Environment Handle
    if (EnvHandle != 0)
        SQLFreeHandle(SQL_HANDLE_ENV, EnvHandle);

    // Return Control To The Operating System
    return(0);
}
```

You may have noticed that a special code named SQL_NTS was passed as a parameter value for some of the CLI/ODBC functions used in this application. CLI/ODBC functions that accept character string values as arguments usually require the length of the character string to be provided as well. The value

SQL_NTS can be used in place of an actual length value to indicate that the corresponding string is null-terminated.

Section 4. Controlling CLI/ODBC driver attributes

Obtaining information about a data source

Because a CLI/ODBC application can connect to a variety of data sources, there may be times when it is necessary to obtain information about a particular data source to which an application is connected. By design, all CLI/ODBC drivers must support three specific functions that, when used, provide information about the capabilities of the driver itself, as well as the capabilities of the driver's underlying data source. Using this set of functions, an application can determine the capabilities and limitations of a particular data source and adjust its behavior accordingly. The first of these functions, the `SQLGetInfo()` function, can be used to obtain information about the various characteristics of a data source. The second function, `SQLGetFunctions()`, tells an application whether or not a particular CLI/ODBC function is supported by a data source/driver. And the last function, `SQLGetTypeInfo()`, provides an application with information about the native data types that are used by a data source. Of the three, `SQLGetInfo()` is probably the most powerful; over 165 different pieces of information can be obtained by this function alone.

The information returned by the `SQLGetInfo()`, `SQLGetFunctions()`, and `SQLGetTypeInfo()` functions is static in nature -- that is, the characteristics of the data source/driver to which the information returned by these three functions refers cannot be altered by the calling application. However, most data source drivers contain additional information that can be changed to alter the way a driver behaves for a particular application. This information is referred to as *driver attributes*. Three types of driver attributes are available:

- Environment attributes
- Connection attributes
- SQL statement attributes

We'll discuss each type in the next few panels.

Environment attributes

Environment attributes affect the way CLI/ODBC functions that operate under a specified environment behave. An application can retrieve the value of an environment attribute at any time by calling the `SQLGetEnvAttr()` function, and it can change the value of an environment attribute by calling the `SQLSetEnvAttr()` function. Some of the more common environment attributes include:

- `SQL_ATTR_ODBC_VERSION`: Determines whether certain functionality exhibits ODBC 2.0 behavior or ODBC 3.x behavior.
- `SQL_ATTR_OUTPUT_NTS`: Determines whether or not the driver is to append

a null terminator to string data values before they are returned to an application.

It is important to note that environment attributes can only be changed as long as no connection handles have been allocated against the environment. Once a connection handle has been allocated, attribute values for that environment can be retrieved, but they cannot be altered.

Connection attributes

Connection attributes affect the way connections to data sources and drivers behave. An application can retrieve the value of a connection attribute at any time by calling the `SQLGetConnectAttr()` function, and it can change the value of a connection attribute by calling the `SQLSetConnectAttr()` function. Some of the more common connection attributes include:

- `SQL_ATTR_AUTOCOMMIT`: Determines whether the data source/driver will operate in autocommit mode or manual commit mode.
- `SQL_ATTR_MAXCONN`: Determines the maximum number of concurrent data source connections that an application can have open at one time. (The default value for this attribute is `0`, which means that an application can have any number of connections open at one time.)
- `SQL_ATTR_TXN_ISOLATION`: Specifies the isolation level to use for the current connection (`SQL_TXN_SERIALIZABLE` for repeatable read, `SQL_TXN_REPEATABLE_READ` for read stability, `SQL_TXN_READ_COMMITTED` for cursor stability, and `SQL_TXN_READ_UNCOMMITTED` for uncommitted read).

As with environment attributes, timing becomes a very important element when setting connection attributes. Some connection attributes can be set at any time, while others can only be set after a corresponding connection handle has been allocated but before a connection to a data source is established. Still other connection attributes can only be set after a connection to a data source is established, while others can only be set after a connection to a data source is established and while there are no outstanding transactions or open cursors associated with the connection.

Statement attributes

Statement attributes affect the way many SQL statement-level CLI/ODBC functions behave. An application can retrieve the value of a statement attribute at any time by calling the `SQLGetStmtAttr()` function, and it can change the value of a statement attribute by calling the `SQLSetStmtAttr()` function. Some of the more common statement attributes include:

- `SQL_ATTR_CONCURRENCY`: Specifies the cursor concurrency level to use (read-only, low-level locking, or value-comparison locking).
- `SQL_ATTR_CURSOR_SENSITIVITY`: Specifies whether cursors on a statement handle are to make changes made to a result data set by another cursor visible. If `SQL_ATTR_CURSOR_SENSITIVITY` is set to `SQL_SENSITIVE`, all cursors on the statement handle make changes made to a result data set by other transactions visible. If this attribute is set to `SQL_INSENSITIVE`, changes made by other transactions will not be seen in the result data set.
- `SQL_ATTR_CURSOR_TYPE`: Specifies the type of cursor that is to be used when processing result data sets (forward-only, static, keyset-driven, or dynamic).
- `SQL_ATTR_RETRIEVE_DATA`: Specifies whether or not the `SQLFetch()` and `SQLFetchScroll()` functions are to automatically retrieve data after they position the cursor.

Once again, timing becomes an important element when setting statement attributes. Some statement attributes must be set before the SQL statement associated with the statement handle is executed, others can be set at any time but are not applied until the SQL statement associated with the statement handle is used again, while other statement attributes can be set at any time.

Section 5. Diagnostics and error handling

Return codes

Each time a CLI/ODBC function is invoked, a special value known as a *return code* is returned to the calling application to indicate whether the function executed as expected. If the function did not execute as expected, the return code value generated will indicate what caused the function to fail. The following table outlines a list of possible return codes that can be returned by any CLI/ODBC function:

Return code	**Meaning**
`SQL_SUCCESS`	The CLI/ODBC function completed successfully.
`SQL_SUCCESS_WITH_INFO`	The CLI/ODBC function completed successfully, however, a warning or non-fatal error condition was encountered.
`SQL_NO_DATA` or `SQL_NO_DATA_FOUND`	The CLI/ODBC function completed successfully, but no relevant data was found.
`SQL_INVALID_HANDLE`	The CLI/ODBC function failed because an invalid environment, connection, statement, or descriptor handle was specified. This return code is only returned when the handle specified either has not been allocated or is the wrong type of handle (for example, if a connection handle is provided when an environment handle is expected). Because this type of error is a programming error, no additional information is provided.
`SQL_NEED_DATA`	The CLI/ODBC function failed because data that the function expected to be available at execution time (such as parameter marker data or connection information) was missing. This return code is typically produced when parameters or columns have been bound as data-at-execution (`SQL_DATA_AT_EXEC`) parameters/columns.
`SQL_STILL_EXECUTING`	A CLI/ODBC function that was started asynchronously is still executing.
`SQL_ERROR`	The CLI/ODBC function failed.

Error handling is an important part of any application, and CLI/ODBC applications are no exception. At a minimum, a CLI/ODBC application should always check to see if a CLI/ODBC function executed successfully by examining the return code produced. Whenever such a function fails to execute as expected, users should be notified that an error or warning condition has occurred. In addition, whenever possible, they should be provided with diagnostic information that will help quickly locate and correct the problem.

SQLSTATEs

Although a return code will notify an application program if an error or warning condition occurred, it does not provide the application (or the developer or a user) with specific information about what caused the error or warning condition. Because additional information about an error or warning condition is usually needed to resolve a problem, DB2 (like other relational database products) uses a set of error message codes known as *SQLSTATEs* to provide supplementary diagnostic information for warnings and errors. SQLSTATEs are alphanumeric strings that are five characters (bytes) in length and have the format *ccsss*, where *cc* indicates the error message class and *sss* indicates the error message subclass. Any SQLSTATE that has a class of `01` corresponds to a warning; any SQLSTATE that has a class of `HY` corresponds to an error that was generated by DB2 CLI; and any SQLSTATE that has a class of `IM` corresponds to an error that was generated by the ODBC Driver Manager. (Because different database servers often have different diagnostic message codes, SQLSTATEs follow standards that are outlined in the X/Open CLI standard specification. This standardization of SQLSTATE values enables application developers to process errors and warnings consistently across different relational database products.)

Unlike return codes, SQLSTATEs are often treated as guidelines, and drivers are not required to return them. Thus, while drivers should always return the proper SQLSTATE for any error or warning they are capable of detecting, applications should not count on this always happening. Because SQLSTATEs are not returned reliably, most applications just display them to the user along with any corresponding diagnostic message and native error code. There is rarely any loss of functionality in taking this approach, because applications normally cannot base programming logic on SQLSTATEs anyway. For example, suppose an application calls the `SQLExecDirect()` function and the SQLSTATE `42000` (Syntax error or access violation) is returned. If the SQL statement that caused this error to occur is hardcoded into the application or constructed at application run time, the error can be attributed to a programming error and the source code will have to be modified. On the other hand, if the SQL statement that caused this error to occur was provided by the user at run time, the error can be attributed to a user mistake, in which case the application has already done all that it can do by informing the user of the problem.

Obtaining diagnostic information

So just how are SQLSTATE values, diagnostic messages, and native error codes obtained when a CLI/ODBC function fails to execute properly? This information is acquired by calling the `SQLGetDiagRec()` function, the `SQLGetDiagField()` function, or both. These functions accept an environment, connection, statement, or descriptor handle as input and return diagnostic information about the last CLI/ODBC function that was executed

using the handle specified. If multiple diagnostic records were generated, an application must call one or both of these functions repeatedly until all diagnostic information available has been obtained. (The total number of diagnostic records available can be determined by calling the `SQLGetDiagField()` function with record number `0` -- the header record number -- and the `SQL_DIAG_NUMBER` option specified.)

Diagnostic information is stored in memory as diagnostic records. Applications can retrieve SQLSTATE values, diagnostic messages, and native error codes from a diagnostic record in a single step by calling the `SQLGetDiagRec()` function. However, this function cannot be used to retrieve information from the diagnostic header record. Instead, applications must use the `SQLGetDiagField()` function to retrieve information stored in the diagnostic header record. The `SQLGetDiagField()` function can also be used to obtain the values of individual diagnostic record fields. (Thus, the `SQLGetDiagField()` function could be used to obtain *just* SQLSTATE values when an error occurs.)

A diagnostic/error handling example

Now that we have seen how return codes can be used to detect when an error or warning condition occurs, and how diagnostic records can be used to provide feedback on how the problem that caused the error/warning to be generated can be corrected, let's see how error handling and diagnostics information retrieval is typically performed in a CLI/ODBC application. A simple CLI/ODBC application written in the C programming language that attempts to connect to a data source using an invalid user ID and that displays the diagnostic information generated when the connection attempt fails might look something like:

```
#include <stdio.h>
#include <string.h>
#include <sqlcli1.h>

int main()
{
    // Declare The Local Memory Variables
    SQLHANDLE    EnvHandle = 0;
    SQLHANDLE    ConHandle = 0;
    SQLRETURN    RetCode = SQL_SUCCESS;

    SQLSMALLINT  Counter = 0;
    SQLINTEGER   NumRecords = 0;
    SQLINTEGER   NativeErr = 0;
    SQLCHAR      SQLState[6];
    SQLCHAR      ErrMsg[255];
    SQLSMALLINT  ErrMsgLen = 0;

    // Allocate An Environment Handle
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE,
        &EnvHandle);

    // Set The ODBC Application Version To 3.x
```

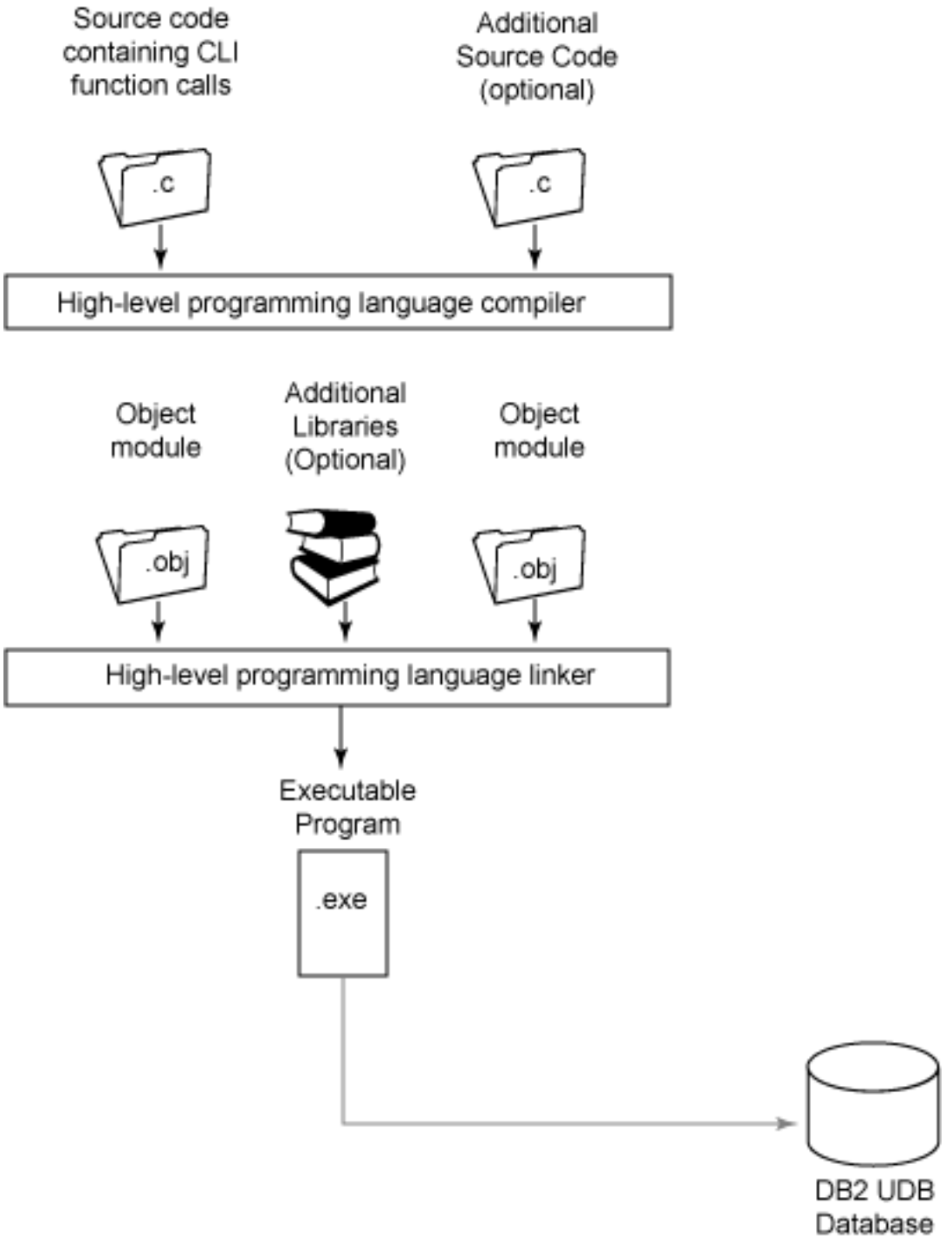
```
    if (EnvHandle != 0)
        SQLSetEnvAttr(EnvHandle, SQL_ATTR_ODBC_VERSION,
            (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);

    // Allocate A Connection Handle
    if (EnvHandle != 0)
        SQLAllocHandle(SQL_HANDLE_DBC, EnvHandle,
            &ConHandle);

    // Attempt To Connect To A Data Source Using An Invalid
    // User ID (This Will Cause An Error To Be Generated)
    if (ConHandle != 0)
        RetCode = SQLConnect(ConHandle, (SQLCHAR *) "SAMPLE",
                      SQL_NTS, (SQLCHAR *) "db2_admin",
                      SQL_NTS, (SQLCHAR *) "ibmdb2",
                      SQL_NTS);

    // If Unable To Establish A Data Source Connection,
    // Obtain Any Diagnostic Information Available
    if (RetCode != SQL_SUCCESS)
    {
        // Find Out How Many Diagnostic Records Are
        // Available
        SQLGetDiagField(SQL_HANDLE_DBC, ConHandle, 0,
            SQL_DIAG_NUMBER, &NumRecords, SQL_IS_INTEGER,
            NULL);

        // Retrieve And Display The Diagnostic Information
        // Produced
        for (Counter = 1; Counter <= NumRecords; Counter++)
        {
            // Retrieve The Information Stored In Each
            // Diagnostic Record Generated
            SQLGetDiagRec(SQL_HANDLE_DBC, ConHandle, Counter,
                SQLState, &NativeErr, ErrMsg, sizeof(ErrMsg),
                &ErrMsgLen);

            // Display The Information Retrieved
            printf("SQLSTATE : %s\n", SQLState);
            printf("%s\n", ErrMsg);
        }
    }

    // Free The Connection Handle
    if (ConHandle != 0)
        SQLFreeHandle(SQL_HANDLE_DBC, ConHandle);

    // Free The Environment Handle
    if (EnvHandle != 0)
        SQLFreeHandle(SQL_HANDLE_ENV, EnvHandle);

    // Return Control To The Operating System
    return(0);
}
```

Section 6. Creating executable applications

Compiling and linking

So far, we have looked at the steps that are used to code CLI/ODBC applications, but we have not seen how the source code for an CLI/ODBC application is converted into a working program. Once a CLI/ODBC source code file has been written, it must be compiled by an appropriate high-level programming language compiler (such as GCC or Visual C++). The compiler is responsible for converting the source code file into an object module that the linker can use to create an executable program. The linker combines object files and high-level programming language libraries to produce an executable application. For most operating systems, this executable application will be an executable module that runs as a stand-alone program. However, it can also be a shared library or a dynamic link library used by another executable module.

The following illustration outlines the basic process for converting CLI/ODBC source code files to an executable application.



It is important to note that DB2 UDB is packaged with a set of special bind files that are used to support DB2 CLI. When a database is created, these files are bound to the database as part of the database creation process to produce a package that facilitates CLI interaction with that database.

Section 7. Conclusion

Summary

This tutorial introduced you to CLI/ODBC programming and to walked you through the basic steps used to construct a CLI/ODBC application. At this point, you should know the difference between environment, connection, statement, and descriptor handles, and you should know how each is used in a CLI/ODBC application. You should also know that CLI/ODBC applications rely on functions to pass SQL statements to a data source for processing, and that these functions must be called in a specific order.

Furthermore, you should know how to establish a database connection from a CLI/ODBC application, how to prepare and execute SQL statements, how to retrieve and process any results produced, and how to terminate transactions. You should also know how to obtain and set driver attributes, how to determine whether or not a CLI/ODBC function executed successfully, and how to obtain diagnostic information if an error occurs.

Finally, you should be familiar with the steps used to convert a source code file containing CLI/ODBC function calls into an executable application.

Resources

For more information on DB2 Universal Database application development:

- *DB2 Version 8 Administration Guide: Implementation*, International Business Machines Corporation, 2002.
- *DB2 Version 8 Application Development Guide: Programming Client Applications*, International Business Machines Corporation, 2002.
- *DB2 Version 8 Application Development Guide: Programming Server Applications*, International Business Machines Corporation, 2002.
- *DB2 Version 8 Application Development Guide: Building and Running Applications*, International Business Machines Corporation, 2002.
- *DB2 Version 8 Call Level Interface Guide and Reference*, International Business Machines Corporation, 2002.

For more information on the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703):

- *DB2 Universal Database v8.1 Certification Exam 703 Study Guide*, Sanders, Roger E., International Business Machines Corporation, 2004.
- *DB2 Universal Database v8 Application Development Certification Guide*, Martineau, David and others, International Business Machines Corporation, 2003.

- *IBM Data Management Skills information* (http://www-4.ibm.com/software/data/db2/skills/)
- *General Certification information* -- including some book suggestions, exam objectives, courses

As mentioned earlier, this tutorial is just one tutorial in a series of seven to help you prepare for the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703). The complete list of all tutorials in this series is provided below:

1. *Database objects and Programming Methods*
2. *Data Manipulation*
3. *Embedded SQL Programming*
4. ODBC/CLI Programming
5. *Java Programming*
6. *Advanced Programming*
7. User-Defined Routines

Before you take the certification exam (DB2 UDB V8.1 Application Development, Exam 703) for which this tutorial was created to help you prepare, you should have already taken and passed the DB2 V8.1 Family Fundamentals certification exam (Exam 700). Use the *DB2 V8.1 Family Fundamentals certification prep tutorial series* to prepare for that exam. A set of six tutorials covers the following topics:

- DB2 planning
- DB2 security
- Accessing DB2 UDB data
- Working with DB2 UDB data
- Working with DB2 UDB objects
- Data concurrency

Use the *DB2 V8.1 Database Administration certification prep tutorial series* to prepare for the DB2 UDB V8.1 for Linux, UNIX and Windows Database Administration certification exam (Exam 701). A set of six tutorials covers the following topics:

- Server management
- Data placement
- Database access
- Monitoring DB2 activity
- DB2 utilities
- Backup and recovery

Check out *developerWorks Subscription* for one-stop access to a comprehensive portfolio of the latest IBM software from DB2, Lotus, Rational, Tivoli, and WebSphere, allowing you to maximize ROI and lower your labor costs, leading to superior productivity.

Feedback

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .